Concertant LLP

Functional Programming Matters

Dr Russel Winder

Partner, Concertant LLP

russel.winder@concertant.com

Copyright © 2008 Russel Winder

Concertant LLP

Functional Programming Matters Not

Dr Russel Winder

Partner, Concertant LLP

russel.winder@concertant.com

Copyright © 2008 Russel Winder

Aims and Objectives of the Session

- Look at how functional programming has influenced and continues to influence imperative/object-oriented programming.
- Consider why functional programming might have relevance in a world that is fundamentally imperative/object-oriented: show how use of C, C++, Java, Python, Ruby, Groovy, etc. is improved by using a more functional programming approach.
- Peek into the increasingly parallel future world and investigate some of the issues of parallelism that functional programming approaches may aid.

Copyright © 2008 Russel Winder

Structure of the Session

- Brief summary of what functional programming is and about some functional programming languages.
- Look at some sample problems, at the techniques for solution, and investigate what is good and what is bad.

This presentation has not been compiled it is being dynamic bound.

Unfortunately, there are no unit tests for this presentation, so errors may arise.

Copyright © 2008 Russel Winder

Background

- 1980s – the declarative vs. imperative war:
  - Imperative: the existing way, but in need of help.
  - Declarative: making the maths of computation excutable.
- The Players:
  - Imperative:
    - Structured programming.
    - Object-oriented programming.
  - Declarative
    - Logic programming.
    - Functional programming.

Copyright © 2008 Russel Winder

Defensiveness and Relevance

- The functional programming community has always been very defensive about real world applicability of functional programming, cf.:

  John Hughes (1984) "Why Functional Programming Matters".

Copyright © 2008 Russel Winder

Hughes' Arguments for FP

- Single assignment.
- No side effects.
- Referential transparency.
- Higher order functions.
- Function composition.
- Newton–Raphson.
- Numerical
  - integration
  - differentiation
- Game trees:
  - Noughts and crosses (aka tic-tac-toe).

Copyright © 2008 Russel Winder

What's FP's Angle?

- Functions.
- Imperative languages do not take functions seriously enough.
  - Obsession with control of mutable state.
  - Obsession with mapping to native machine architectures.

Copyright © 2008 Russel Winder

Current Players in the Functional Programming Arena

- **Scheme** – a variant of Lisp.
- **OCaml** – the object-oriented form of ML and Caml.
- **Haskell** – the descendant of ML, Hope, KRC, Miranda.
- **Erlang** – the functional form of Occam.

There is a sort of date ordering here:
Lisp precedes ML, which precedes Miranda.

Occam is definitely not a functional programming language

Copyright © 2008 Russel Winder

On Expertise

- Expertise is *not* the number of years of experience in a given language – though that is a factor.
- Expertise is strongly related to the number of different types of language a person is fluent in.

Learning and being able to use C, C++, Java, Python, Ruby, Groovy, Fortran, Haskell, Erlang, etc. ***properly*** *makes you a better programmer.*

Copyright © 2008 Russel Winder

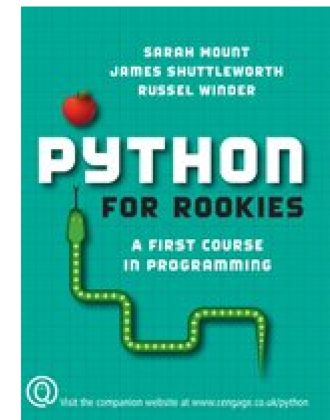
Subliminal Advertising

Python for Rookies

Sarah Mount, James Shuttleworth and

Russel Winder

Thomson Learning



Now called Cengage Learning.

Learners of Python need this book.

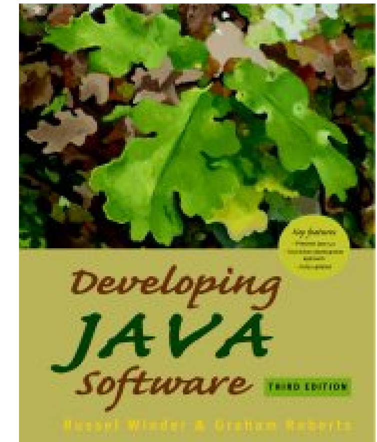
More Subliminal Advertising

Developing Java Software

Third Edition

Russel Winder and Graham Roberts

Wiley



Learners of Java need this book.

Copyright © 2008 Russel Winder

Anti Advertising

Developing C++ Software

Second Edition

Russel Winder

Wiley

Only buy this book is you are studying the history of C++ and how it was taught.

Learners of C++ used to need this book.

Copyright © 2008 Russel Winder

What is Functional Programming?

- Computation by transformation of data using function application.
- The features:
  - Referential transparency of function calls.
  - No side-effects, no mutable data.
  - Functions as first class entities; higher-order functions.

On this Hughes is 100% correct.

Copyright © 2008 Russel Winder

Crucial Change of Perspective

- Imperative:
  - Procedures and methods undertake some activity possibly resulting in a return value.
- Functional:
  - Data is transformed by the application of referentially transparent functions.

The realization of algorithm is handled very differently.

Copyright © 2008 Russel Winder

Distinguishing Functional Programming Languages

- Strict / Non-strict
- Eager / Lazy
- Pure / Impure

Copyright © 2008 Russel Winder

Fibonacci Numbers
Introduction

- The Fibonacci Sequence is the sequence of numbers defined by the recurrence relation:

$$F(n) = \begin{cases} 0 & n=0 \\ 1 & n=1 \\ F(n-1)+F(n-2) & n>1 \end{cases}$$

Copyright © 2008 Russel Winder

Fibonacci Numbers
Naïve Implementation

- People are generally tempted to implement the function directly from the definition as recursive functions.
- The following show such implementations in C, C++, Java, Python, Groovy, Haskell, and Erlang.

Copyright © 2008 Russel Winder

Fibonacci Numbers
Naïve Static Imperative

```
unsigned long long naiveRecursive ( const unsigned long long n ) {
  return n == 0 ? 0 : n == 1 ? 1 : naiveRecursive ( n - 1 ) + naiveRecursive ( n – 2 ) ;
}
```

```
public static BigInteger naiveRecursive ( final  long n ) {
  return
    n == 0 ? BigInteger.ZERO :
    n == 1 ? BigInteger.ONE : naiveRecursive ( n - 1 ).add ( naiveRecursive ( n - 2 ) ) ;
}
```

Copyright © 2008 Russel Winder

Fibonacci Numbers
Naïve Dynamic Imperative

```
def naiveRecursive ( n ) :
   return 0 if n == 0 else 1 if n == 1 else naiveRecursive ( n -1 ) + naiveRecursive ( n - 2 )
```

```
def naiveRecursive ( n ) {
  return n == 0 ? 0 : n ==1 ? 1 : naiveRecursive ( n - 1 ) + naiveRecursive ( n - 2 )
}
```

Copyright © 2008 Russel Winder

Concertant LLP

Fibonacci Numbers
Naïve Functional

```
naiveRecursivePattern :: Integer -> Integer
naiveRecursivePattern 0 = 0
naiveRecursivePattern 1 = 1
naiveRecursivePattern n = naiveRecursivePattern ( n - 1 ) + naiveRecursivePattern ( n - 2 )

naiveRecursiveGuarded :: Integer -> Integer
naiveRecursiveGuarded n
    | n == 0 = 0
    | n == 1 = 1
    | n > 1 = naiveRecursiveGuarded ( n - 1 ) + naiveRecursiveGuarded ( n - 2 )
```

```
naiveRecursive ( 0 ) -> 0 ;
naiveRecursive ( 1 ) -> 1 ;
naiveRecursive ( N ) when N > 1 -> naiveRecursive ( N - 1 ) + naiveRecursive ( N - 2 ) ;
naiveRecursive ( N ) when N < 0 -> erlang:error ( { fibonacciNaiveRecursiveNegativeArgument , N } ) .
```

Copyright © 2008 Russel Winder

Fibonacci Numbers Haskell, Better Pattern Matching Version?

```
naiveRecursiveBetterPattern :: Integer -> Integer
naiveRecursiveBetterPattern 0 = 0
naiveRecursiveBetterPattern 1 = 1
naiveRecursiveBetterPattern ( n + 2 ) =
    naiveRecursiveBetterPattern ( n + 1 ) + naiveRecursiveBetterPattern n
```

Copyright © 2008 Russel Winder

Fibonacci Numbers
A Trick of the Tail (Recursion)

The Genesis of sophistication.

```
tailRecursiveGuarded :: Integer -> Integer
tailRecursiveGuarded n
    | n < 0 = error "Cannot use negative parameter."
    | n == 0 = 0
    | n == 1 = 1
    | n > 1 = doTailRecursion n 1 0
      where
        doTailRecursion n next result = if n == 0
                                        then result
                                        else doTailRecursion ( n - 1 ) ( next + result ) next
```

Copyright © 2008 Russel Winder

Fibonacci Numbers
A Trick of the Tail (Recursion)

You can (Tony) Banks on it.

```
tailRecursive ( 0 ) -> 0 ;
tailRecursive ( 1 ) -> 1 ;
tailRecursive ( N ) when N > 0 -> iterateTailRecursive ( 1 , N , 1 , 0 ) ;
tailRecursive ( N ) when N < 0 ->
    erlang:error ( { fibonacciTailRecursiveNegativeArgument , N } ) .

iterateTailRecursive ( N , Max , Next , Result ) when N > Max ->  Result ;
iterateTailRecursive ( N , Max , Next , Result ) ->
    iterateTailRecursive ( N + 1 , Max , Next + Result , Next ) .
```

Copyright © 2008 Russel Winder

Concertant LLP

Fibonacci Number
A Trick of the Tail (Recursion)

During the session Jeremy Gibbons suggested having these two versions made public.

```
tailRecursivePattern :: Integer -> Integer
tailRecursivePattern 0 = 0
tailRecursivePattern 1 = 1
tailRecursivePattern ( n + 2 ) = doTailRecursion ( n + 2 ) 0 1 where
    doTailRecursion 0 result next = result
    doTailRecursion ( n + 1 ) result next = doTailRecursion n next ( result + next )
```

```
tuplingTrick :: Integer -> Integer
tuplingTrick n = fst ( doDouble n ) where
    doDouble 0 = ( 0 , 1 )
    doDouble ( n + 1 ) = let ( a , b ) = doDouble n in ( b , a + b )
```

Copyright © 2008 Russel Winder

Fibonacci Numbers
Some Observations on Being Naïve

- The implementations are using a function to calculate a value in a sequence – should really just use a data structure to represent a sequence.
- Infinite sequences cannot be represented as data structures, so use a function.
- The naïve implementation is fundamentally unsound – $O(2^n)$ in fact.
- MUST use memoization to avoid replication of calculations if using a recursive implementation.

Copyright © 2008 Russel Winder

Concertant LLP

Fibonacci Numbers
Imperatively, Statically Iterative

```
unsigned long long iterative ( const unsigned long long n ) {
  unsigned long long result = 0 ;
  unsigned long long next = 1 ;
  for ( unsigned long long i = 0 ; i < n ; ++i ) {
    unsigned long long temporary = result ;
    result = next ;
    next += temporary ;
  }
  return result ;
}
```

```
public static BigInteger iterative ( final long n ) {
  BigInteger result = BigInteger.ZERO ;
  BigInteger next = BigInteger.ONE ;
  for ( long i = 0 ; i < n ; ++i ) {
    BigInteger temporary = result ;
    result = next ;
    next = temporary.add ( next ) ;
  }
  return result ;
}
```

Copyright © 2008 Russel Winder

Fibonacci Numbers Imperatively, Dynamically Iterative

```
def iterative ( n ) :
    result = 0
    next = 1
    for i in range ( n ) :
        temporary = result
        result = next
        next = temporary + next
    return result
```

```
def iterative ( n ) {
  def result = 0
  def next = 1
  for ( i in 0..<n ) {
    def temporary = result
    result = next
    next += temporary
  }
  result
}
```

Copyright © 2008 Russel Winder

Fibonacci Numbers
But Recursive Looks Better

- The iterative solution certainly performs faster, O(n), than the naïve recursive solution, O(2^n).
- It would be better to do better.
- Use memoization to build a data structure – get to O(q)

Copyright © 2008 Russel Winder

Fibonacci Numbers
Keeping a C++ Memo

```
unsigned long long memoizedRecursive ( const unsigned long long n ) {
  static std::map<unsigned long long,unsigned long long> memo ;
  if ( memo.count ( n ) == 0 ) {
    switch ( n ) {
     case 0 : memo[0] = 0 ; break ;
     case 1 : memo[1] = 1 ; break ;
     default : memo[n] = memoizedRecursive ( n - 1 ) + memoizedRecursive ( n - 2 ) ; break ;
    }
  }
  return memo[n] ;
}
```

Copyright © 2008 Russel Winder

Concertant LLP

Fibonacci Numbers
Keeping a Java Memo

```
static Map<Long,BigInteger> memo = new HashMap<Long,BigInteger> ( ) ;
static {
  memo.put ( 0l , BigInteger.ZERO ) ;
  memo.put ( 1l , BigInteger.ONE ) ;
}
public static BigInteger memoizedRecursive ( final long n ) {
  if ( ! memo.containsKey ( n ) ) {
    memo.put ( n , memoizedRecursive ( n - 1 ).add ( memoizedRecursive ( n - 2 ) ) ) ;
  }
  return memo.get ( n ) ;
}
```

Copyright © 2008 Russel Winder

Concertant LLP

Fibonacci Numbers
Keeping a Dynamic Memo

```
memo = { 0 : 0 , 1 : 1 }
def memoizedRecursive ( n ) :
    if not n in memo :
        memo[n] = memoizedRecursive ( n - 1 ) + memoizedRecursive ( n - 2 )
    return memo[n]
```

```
def memoizedRecursive ( n ) {
  static memo = [ 0 : 0 , 1 : 1 ]
  if ( ! memo.containsKey ( n ) ) {
    memo[n] = memoizedRecursive ( n - 1 ) + memoizedRecursive ( n - 2 )
  }
  memo[n]
}
```

Copyright © 2008 Russel Winder

Fibonacci Numbers
Being Lazy but Functional

```
lazyList :: Integer -> Integer
lazyList n = fibonacciSequence !! fromInteger n
  where
    fibonacciSequence = 0 : 1 : zipWith (+) fibonacciSequence ( tail fibonacciSequence )
```

Lazy is good.

Copyright © 2008 Russel Winder

Historical Factors

- Functional programming was the preserve of the computer science theory people.
- Functional programming seen as too academic since it was espoused by academics.
- "Real Programmers use Imperative Languages."

Copyright © 2008 Russel Winder

Higher Order Functions

- C and C++ have function pointers but this is nothing like as powerful as the feature of higher order functions.
- Dynamic languages (e.g. Python, Groovy, Ruby) have taken on board functions as first class entities and hence higher order functions.

Copyright © 2008 Russel Winder

Python as Anti-Pattern

- Guido van Rossum has publicly stated that reduce is incomprehensible – summation is the only use case.
- The functional languages and all the other main dynamic languages take the opposite view.
- Reduce can be complicated to use, but then so are higher order functions in general unless you are careful.

Copyright © 2008 Russel Winder

Erlang – System Programming the Functional Way

- Erlang is a functional programming language, yet it is used to write telecoms switches!
- Supports parallelism – process (not thread!) is the primitive unit of parallelism.

Copyright © 2008 Russel Winder

Scala – An Interesting Language?

- Scala tries to merge object-oriented and functional programming in a single language.

Scala has integral preconditions.

Copyright © 2008 Russel Winder

C++ Requires a Functional Approach

- C++ encourages a functional programming approach with the STL and generic functions.
  - Learning functional programming helps with idioms and strategies of C++ programming.
- C++ requires a functional approach for template programming.
  - Learning functional programming gives you a handle on doing template metaprogramming.

Copyright © 2008 Russel Winder

Summary

- Functional programming may never be the dominant paradigm.
- Haskell and Erlang are used and being used more in real applications.
- The way of programming in functional languages is applicable to other languages, e.g. C, C++, Java as well as Python, Groovy, Ruby, Perl, etc.

Copyright © 2008 Russel Winder

One Liner

- Knowing how to program in a functional language helps programming in any and all other languages.

Copyright © 2008 Russel Winder